Coursework 1 - Big Data

Student: Ilias Papounidis
Student ID: 140422715

Part A: Message Length Analysis

Mapper

The mapper in this situation tries to divide the number by five so that it can be easily group-able in groups that are of interval 5. The `Math.ceil(valueFloat)` statement just makes the value that is divided become its closest integer value.

Location: part1/TweeterSizeMapper.java

```
 public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
    String[] line = value.toString().split(";");
    /*
      Each line is devided into four parts:
            0       1               2                  3
      epoch_time;tweetId;tweet(including #hashtags);device
    */
    if (line.length == 4) {

      int length = line[2].length();
      if (length <= 140) {
        context.write(group(length), one);
      }
    }
  }
  // Takes an input an integer and makes it work with the Reducer to put each
key into a group
  public IntWritable group(int value) {

    float valueFloat = (float) value/5;
    int group = (int) Math.ceil(valueFloat);

    return new IntWritable(group);
  }
```

Reducer

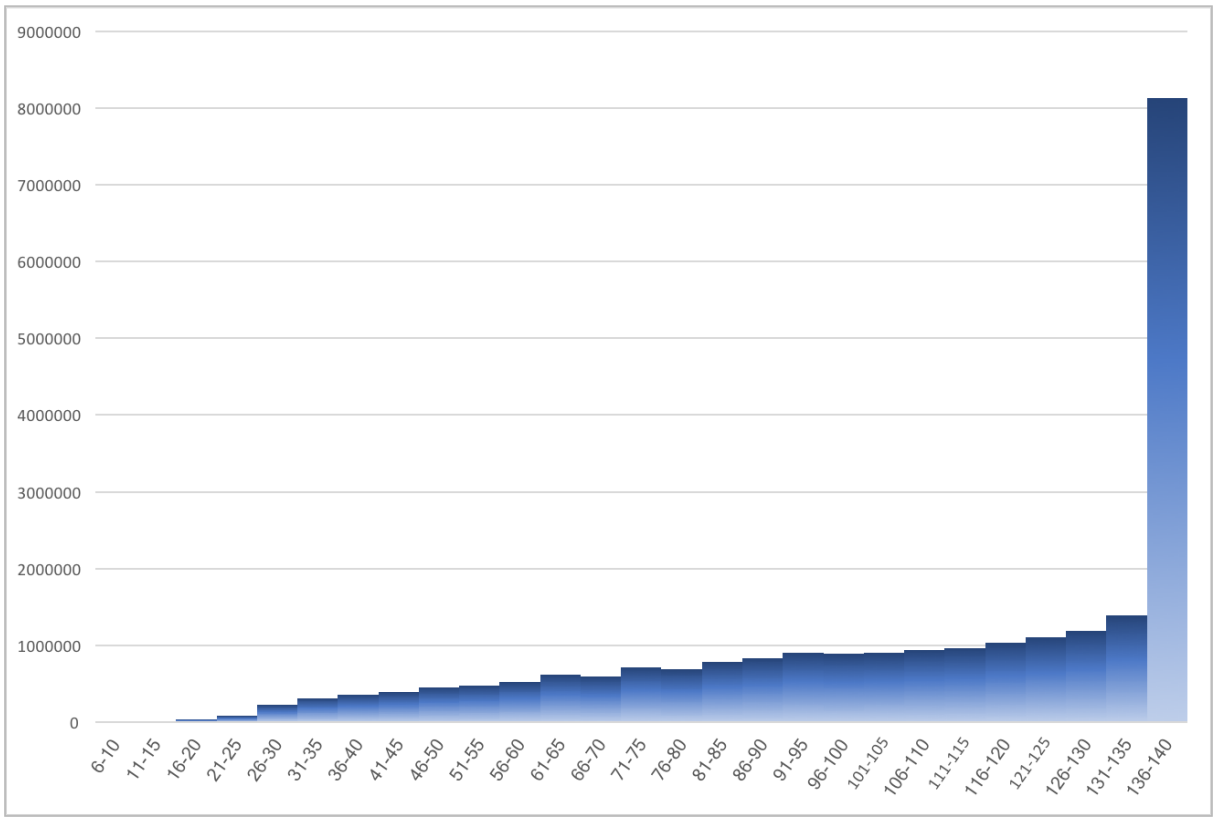
The reducer in the implementation takes the values from the mapper and manages to put it in the right group with this statement: `Text((key.get()*5-4)+"-"+(key.get()*5));`

So in the end the reducer emits the count of the size in intervals of 1-5.

> Location: part1/TweeterSizeReducer.java

```
public void reduce(IntWritable key, Iterable<IntWritable> values, Context
context)
              throws IOException, InterruptedException {

    int sum = 0;

    for (IntWritable value : values) {
      sum = sum + value.get();
    }

    result.set(sum);
    Text finalKey =new Text((key.get()*5-4)+"-"+(key.get()*5));
    context.write(finalKey, result);

}
```

Histogram



As we can se from the above histogram, the most popular tweets are in the range 136-140.

This is all calculated from the csv file `count1.csv`.

Part B: Time Analysis <Part 1>

Mapper

In this implementation we take in the mapper as input the tweet and we split it. After that we take the epoch value from the tweet and parse it as a `long` variable. Finally we create a `LocalDateTime` object and use it to output the Hour value through `context.write(hour, one)`. Additionally we correct the time by applying `ZoneOffset.of("-2")` in the LocalDateTime object.

> Location: part2/TweetTimeMapper.java

```
public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {

    String[] line = value.toString().split(";");
    /*
      Each line is devided into four parts:
            0       1               2                  3
      epoch_time;tweetId;tweet(including #hashtags);device
    */
    if (line.length == 4) {
      try {
        long epoch_time = Long.parseLong(line[0]);
        LocalDateTime time = LocalDateTime.ofEpochSecond(epoch_time/
1000,0,ZoneOffset.of("-2"));


        IntWritable timeGroup = new IntWritable(time.getHour());
        context.write(timeGroup,one);
      } catch (Exception e) {}
    }
  }
}
```

Reducer

The reducer here just takes and calculates the number of appearances of this hour in the tweets. Like this we are able to compute the hour that was most popular.

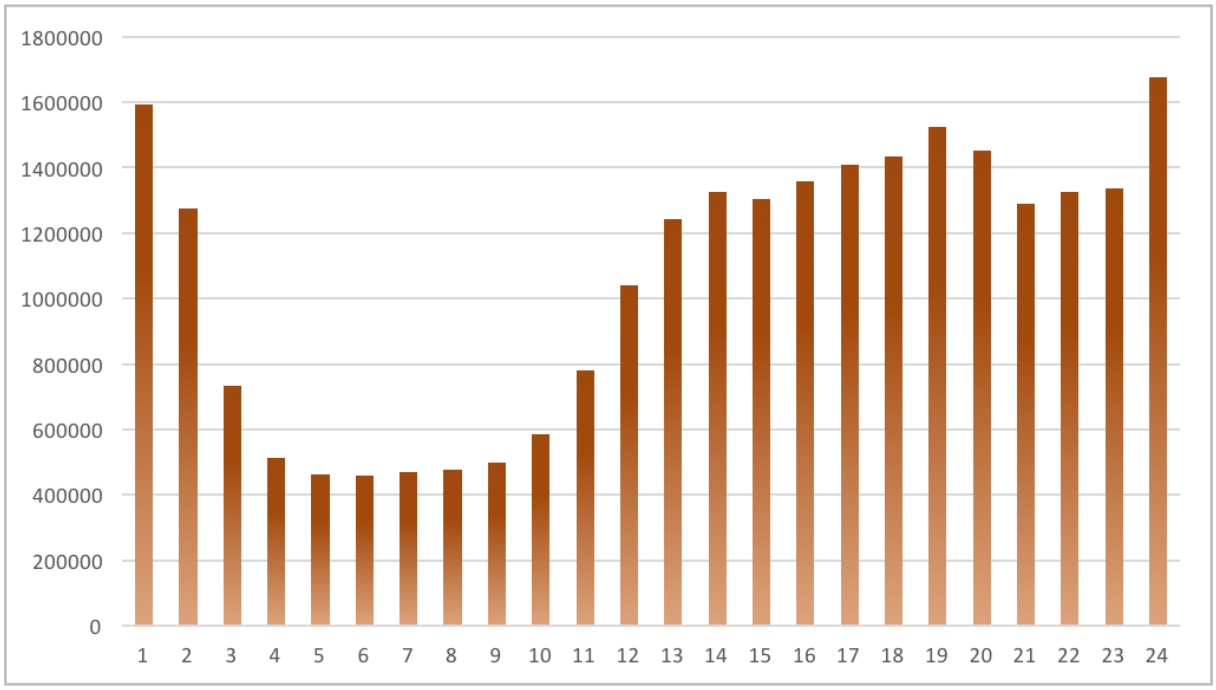
> Location: part2/TweetTimeReducer.java

```
  private IntWritable result = new IntWritable();

  public void reduce(IntWritable key, Iterable<IntWritable> values, Context
context)
              throws IOException, InterruptedException {
```

```
    int sum = 0;

    for (IntWritable value : values) {
      sum = sum + value.get();
    }

    result.set(sum);
    Text finalKey = new Text("Hour: " + key.toString());
    context.write(finalKey, result);
  }
```

Bar plot

From the output file count2 after sorting it in terminal with the command `sort -rn -k2 count2` we can easily see that the most popular hour is 23h. Here is a representation of the bar plot:



Part B: Time Analysis <Part 2>

Using the information from <Part 1> we realise that the most popular hour was at 23h. We use that information and we hardcode it to the new mapreduce to calculate the most popular hashtags.

Mapper

The mapper here uses an if statement to check if the tweet was made at the most popular hour. We later split the tweet into words that are separated by spaces. Then, we initialise a for lop that goes through the tweet and checks if there is a hashtag inside. As soon as the program finds a hashtag it emits a `context.write()` function with the hashtag included. The last if statement uses the regular expression `"[##]+([A-Za-z0-9-_]+)"` to realise if a word is a hashtag or not.

Location: part2b/PopHashtagMapper.java

```
  public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {

    String[] line = value.toString().split(";");
    /*Each line is devided into four parts:
            0       1               2                   3
      epoch_time;tweetId;tweet(including #hashtags);device
    */
    if (line.length == 4) {
      try {
        long epoch_time = Long.parseLong(line[0]);
        LocalDateTime time = LocalDateTime.ofEpochSecond(epoch_time/
1000,0,ZoneOffset.of("-2"));

        if (time.getHour() == 23) {
          String[] tweet = line[2].split(" ");
          for (String word : tweet) {
            if(word.matches("[##]+([A-Za-z0-9-_]+)")) {
            //I have tried not limiting it only to english characters and
numbers and the output is the same
              context.write(new Text(word),one);
            }
          }
        }
      } catch (Exception e) {}
    }
  }
```

Reducer

The reducer here is very similar to the previous implementations and it just counts the amount of appearances of each hashtag.

Location: part2b/PophashtagReducer.java

```
public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {

  int sum = 0;
  for (IntWritable value : values) {
    sum = sum + value.get();
  }
  result.set(sum);
  context.write(key, result);
}
```

Output

Here the requested output is a list of the most popular hashtags calculated by the program. The program only calculated the number of appearances of each hashtag. By running a command in terminal we can easily pipe that output to a sorted list with the top hashtags using: `sort -rn -k2 count2b | head -n 10`

```
#Rio2016  1267277
#Olympics 74903
#rio2016  70193
#Futebol  39751
#Gold 37266
#BRA  35045
#USA  34431
#CerimoniaDeAbertura  33365
#OpeningCeremony  32574
#Atletismo 28717
```

The above is the aggregated list of the top hashtags.

Part C: Support Analysis <Part 1>

The goal of the final part is to compute the 30 athletes that received the highest support, according to the Twitter messages of the dataset. The way this is done is by using an external csv file with the names of the athletes and then searching them in each tweet. There are some changes here to how the job part is done so the code will be included here as well.

Job

For this question we needed to import an external file during the run of our job. In order for us to do that we have to execute the command: `job.addCacheFile(new Path("../../data/medalistsrio.csv").toUri());`

Mapper

Here the mapper had to implement extra features. Specifically here we needed a side join of the data from the csv file using a HashSet. There is a part in the code that was very heavily inspired by the code given during Lab4 as it helped do exactly the thing we needed.

As it can be seen in the code bellow the implementation done in this coursework uses a `HashSet` to store the values that have been found in the csv file. The program continues to iterate through the values of the `HashSet` to find matches of the names in the set with the names used in the tweets.

This implementation is not the most efficient one possible but it serves the purpose of easily finding names with several middle names or even names that consist of just one word.

> Location: part3/TweetCountJoinMapper.java

```
  public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {

          String[] tweetValues = value.toString().split(";"); //Split value
into tweet values
          if (tweetValues.length > 2) {
              String tweet = tweetValues[2];
```

```
                Iterator itr2 = athleteSet.iterator();
                while (itr2.hasNext()) {
                    String name = itr2.next().toString();
                    if(tweet.contains(name)) {
                        context.write(new Text(name), one);
                    }
                }
            }
    }
```

The setup method:

```
    @Override
    protected void setup(Context context) throws IOException,
InterruptedException {

        athleteSet = new HashSet<String>();

        // We know there is only one cache file, so we only retrieve that URI
        URI fileUri = context.getCacheFiles()[0];

        FileSystem fs = FileSystem.get(context.getConfiguration());
        FSDataInputStream in = fs.open(new Path(fileUri));

        BufferedReader br = new BufferedReader(new InputStreamReader(in));
        String line = null;
        try {
            // we discard the header row
            br.readLine();

            while ((line = br.readLine()) != null) {

                String[] fields = line.split(",");
                if (fields.length == 11) {
                    //fields are: 0:ID 1:Full Name 2:Country 3:Sex 4:Date of
birth 5:Height 6:Weight 7:Sport 8:Gold Medals 9:Silver Medals 10:Bronze Medals
```

```
                    athleteSet.add(" " + fields[1]);
                }
            }
            br.close();
        } catch (IOException e1) {
        }

        super.setup(context);
    }
```

Reducer

The reducer here again doesn't do much different from the ones we've seen before and it just helps with counting the amount of mentions of each athlete.

Location: part3/TweetCountReducer.java

Output

In order for us to fin the top athletes we need to sort the outputted data and just show the top 30 people. The way to do this is to run this command in terminal: `awk '{print $NF, $0}' count3 | sort -nr | cut -f2- -d' ' | head -n 30`

What it does is that it switches the values from the last column (the column that actually has the mentions) to the first column so the data can be easily sorted by the numbers.

The outputted table is as follows: (next page)

Name	Mentions
Michael Phelps	16903
Usain Bolt	15649
Neymar	7928
Simone Biles	7227
Ryan Lochte	3818
William	3483
Katie Ledecky	3291
Yulimar Rojas	3256
Joseph Schooling	2571
Simone Manuel	2562
Sakshi Malik	2383
Rafaela Silva	2144
Tontowi Ahmad	2138
Kevin Durant	2044
Andy Murray	1972
Penny Oleksiak	1868
Wayde van Niekerk	1757
Monica Puig	1567
Laura Trott	1469
Rafael Nadal	1406
Ruth Beitia	1368
Teddy Riner	1318
Shaunae Miller	1163
Lilly King	1128
Elaine Thompson	1103
Hidilyn Diaz	1098
Caster Semenya	1047
Almaz Ayana	1040
Jason Kenny	1033
Allyson Felix	1016

Part C: Support Analysis <Part 2>

This part was asking for the top 20 mentions by sport based on the mentions of the athletes. This can be done using the output of the previous mapreduce job file `count3` and using it as an input for this job. Additionally this job will need the information from the `medalistsrio.csv` file to connect the names of the athletes with their according sport.

Mapper

The mapper here is very similar to the previous job, however it now uses a HashTable to store the according athlete information. This makes it easier for us to access the sport of an athlete with just one command: `athletesTable.get(name).toString();`.
As we can see bellow the names of the athletes needed to be trimmed from trailing spaces otherwise the above function wasn't able to find them. The emitted values are of the kind `fencing 23 / canoe 35 / fencing 32 / football 7928`

> Location: part3b/TweetCountJoinMapper.java

```
  public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {

          String[] values = value.toString().split("\t");

          if (values.length == 2) {

              try {
                  if (values[0] != null && values[1] != null) {
                      String name = values[0].trim();
                      String sport = athletesTable.get(name).toString();
                      int mentions = Integer.parseInt(values[1]);
                      textSport.set(new
Text(athletesTable.get(name).toString()));
                      context.write(textSport, new IntWritable(mentions));
                  }
              } catch (Exception e) {}
          }
  }
```

The setup method:

As we can see here the setup method puts the values to the `Hashtable` and makes sure all the values inside are not null.

```
  @Override
  protected void setup(Context context) throws IOException,
InterruptedException {

      athletesTable = new Hashtable<String,String>();

      // We know there is only one cache file, so we only retrieve that URI
      URI fileUri = context.getCacheFiles()[0];

      FileSystem fs = FileSystem.get(context.getConfiguration());
      FSDataInputStream in = fs.open(new Path(fileUri));

      BufferedReader br = new BufferedReader(new InputStreamReader(in));
      String line = null;
      try {
          // we discard the header row
          br.readLine();
          while ((line = br.readLine()) != null) {
              String[] fields = line.split(",");
              if (fields.length > 7) {
                  if ((fields[1]!=null) && (fields[7]!=null)){
                      //fields are: 0:ID 1:Full Name 2:Country 3:Sex 4:Date of
birth 5:Height 6:Weight 7:Sport 8:Gold Medals 9:Silver Medals 10:Bronze Medals
                      athletesTable.put(fields[1],fields[7]);
                  }
              }
          }
          br.close();
      } catch (IOException e1) {
      }
      super.setup(context);
  }
}
```

Reducer

The reducer here again doesn't do much different from the ones we've seen before and it just helps with counting the amount of mentions for each sport based on the emitted values of the mapper.

> Location: part3b/SportCountReducer.java

Output

As requested the output here is a table of the top sports based on athlete mentions. The output file was then sorted and cut to just the top 20 values with the command: `sort -rn -k2 out/count3b >> count3b_top20 | head -n 20`

This is what the code managed to calculate: (next page)

Sport	Mentions
athletics	42286
aquatics	41666
football	16150
gymnastics	11675
judo	9192
tennis	7277
basketball	6693
cycling	6232
badminton	4226
wrestling	3280
sailing	2333
weightlifting	2303
canoe	2293
shooting	2174
equestrian	2167
boxing	2152
rowing	1699
volleyball	1570
taekwondo	1484
fencing	1148